

**VIRTUALIZING SUPER-USER PRIVILEGES FOR  
MULTIPLE VIRTUAL PROCESSES**

**INVENTORS**

Xun Wilson Huang, Cristian Estan, and Srinivasan Keshav

Prepared by:

Fenwick & West LLP  
Two Palo Alto Square  
Palo Alto, CA 94306

Express Mail No.: EL482716691US

# VIRTUALIZING SUPER-USER PRIVILEGES FOR MULTIPLE VIRTUAL PROCESSES

## INVENTORS

Xun Wilson Huang, Cristian Estan, and Srinivasan Keshav

## BACKGROUND

### FIELD OF THE INVENTION

The present invention relates generally to computer operating systems, and more particularly, to techniques for virtualizing super-user privileges in a computer operating system including multiple virtual processes, such as virtual private servers.

### DESCRIPTION OF THE BACKGROUND ART

With the popularity and success of the Internet, server technologies are of great commercial importance today. An individual server application typically executes on a single physical host computer, servicing client requests. However, providing a unique physical host for each server application is expensive and inefficient.

For example, commercial hosting services are often provided by an Internet Service Provider (ISP), which generally provides a separate physical host computer for each customer on which to execute a server application. However, a customer purchasing hosting services will often neither require nor be amenable to paying for use

of an entire host computer. In general, an individual customer will only require a fraction of the processing power, storage, and other resources of a host computer.

Accordingly, hosting multiple server applications on a single physical computer would be desirable. In order to be commercially viable, however, every server application would need to be isolated from every other server application running on the same physical host. Clearly, it would be unacceptable to customers of an ISP to purchase hosting services, only to have another server application program (perhaps belonging to a competitor) access the customer's data and client requests. Thus, each server application program needs to be isolated, receiving requests only from its own clients, transmitting data only to its own clients, and being prevented from accessing data associated with other server applications.

Furthermore, it is desirable to allocate varying specific levels of system resources to different server applications, depending upon the needs of, and amounts paid by, the various customers of the ISP. In effect, each server application needs to be a "virtual private server," simulating a server application executing on a dedicated physical host computer.

Such functionality is unavailable on traditional server technology because, rather than comprising a single, discrete process, a virtual private server must include a plurality of seemingly unrelated processes, each performing various elements of the sum total of the functionality required by the customer. Because each virtual private server includes a plurality of processes, it has been impossible using traditional server

technology for an ISP to isolate the processes associated with one virtual private server from those processes associated with other virtual private servers.

Accordingly, what is needed is a technique for associating a plurality of processes with a virtual process. What is also needed is a technique for associating an identifier with a virtual process.

20.7 One of the difficulties in providing isolation between virtual private servers within a single host computer involves resource ownership. In UNIX® and related operating systems, certain system resources, such as processes and files, are owned by users or group of users. Each user is assigned a user identifier (UID) by which the user is identified in the operating system. In some cases, a group of users may be assigned a group identifier (GID).

Resource ownership is typically used to implement access control. For example, a user can generally only kill a process or access a file that he or she owns (or for which permission has been granted by the owner). Thus, if a user attempts, for instance, to kill a process that he or she does not own, the attempt fails and an error is generated.

An exception to the above is a special user, known as a "super-" or "root-" user. The super-user has access to all system resources and is typically a system administrator or the like. For example, the super-user can open, modify, or delete any system file and can terminate any system process.

20 Implementing resource ownership in the context of multiple virtual private servers presents a number of difficulties. Each virtual private server should be free to

assign to an individual or group any UID or GID, respectively. Indeed, some applications require certain files or processes to be associated with a particular UID or GID in order to properly function.

Unfortunately, if two users of different virtual private servers share the same UID, one user could potentially kill the other user's processes and read, modify, or delete the other user's files. The same possibility is true for two groups sharing the same GID.

For example, one user could execute a "kill -1" command, which terminates all of the processes associated with the user's UID. Unfortunately, if another user on the same computer shares the same UID, all of that user's processes will be terminated as well. Clearly, this is unacceptable in the context of a virtual private server, where each server should appear to be running on a dedicated host machine.

Accordingly, what is needed is a technique for virtualizing resource ownership in a computer operating system including multiple virtual private servers. Indeed, what is needed is a technique for allowing a virtual private server to assign any UID or GID to a user or group, without creating an unacceptable security risk or removing the appearance that the virtual private server is running on a dedicated host.

As noted above, in UNIX® and related operating systems, the super-user is granted special privileges not available to other users. For example, the super-user can open, modify, or delete the files of other users, as well as terminate other users'

processes. Indeed, the super-user can add and delete users, assign and change passwords, and insert modules into the operating system kernel.

Implementing super-user privileges in a computer operating system including multiple virtual processes presents numerous difficulties. For example, each virtual process should be allowed to have a system administrator who has many of the privileges of a super-user, e.g., the ability to add and delete users of the virtual process, access files of any user of the virtual process, terminate processes associated with the virtual process, and the like.

However, if a user of each virtual process was given full super-user privileges, a super-user of one virtual process could access the files of a user of another virtual process. Similarly, a super-user of one virtual process could terminate the processes associated with a user of another virtual process. Indeed, a super-user of one virtual process could obtain exclusive access to all system resources, effectively disabling the other virtual processes. Clearly, allowing a user of each virtual process full super-user privileges would seriously compromise system security, entirely removing the illusion that the virtual processes are running on dedicated host computers.

Accordingly, what is needed is a technique for virtualizing super-user privileges in a computer operating system including multiple virtual processes. Moreover, what is needed is a technique for virtualizing super-user privileges, such that a virtual super-user has the power to perform traditional system administrator functions with respect



## SUMMARY OF THE INVENTION

The present invention relates to virtualizing super-user privileges in a computer operating system including multiple virtual processes. In one aspect of the invention, a plurality of virtual super-users are designated, each virtual super-user being associated with a separate virtual process. A virtual super-user may be designated, in one embodiment, by assigning a virtual super-user identifier, which may comprise a super-user identifier and an indication of a virtual process. In an alternative embodiment, a virtual super-user may be designated by assigning a regular user identifier and storing that identifier in a virtual super-user list.

In another aspect of the invention, a system call wrapper intercepts a system call for which actual super-user privileges are required, which is nevertheless desirable for a virtual super-user to perform in the context of his or her own virtual process. In response to a determination that the intercepted system call was made by a virtual super-user and pertains to the virtual process of the virtual super-user, the virtual super-user is temporarily granted actual super-user privileges. The system call is then executed as though it were made by real super-user, after which the actual super-user privileges are withdrawn.

Thus, a virtual super-user has the power to perform traditional system administrator functions with respect to his or her own virtual process, but is unable to interfere with other virtual processes or the underlying operating system. Moreover,



each virtual process may have a virtual super-user, while preserving the illusion that the virtual processes are running on dedicated host machines.

The features and advantages described in this summary and the following detailed description are not all-inclusive, and particularly, many additional features and advantages will be apparent to one of ordinary skill in the art in view of the drawings, specification, and claims hereof. Moreover, it should be noted that the language used in the specification has been principally selected for readability and instructional purposes, and may not have been selected to delineate or circumscribe the inventive subject matter, resort to the claims being necessary to determine such inventive subject matter.

## BRIEF DESCRIPTION OF THE DRAWINGS

FIG. 1 is a block diagram of a system for associating identifiers with virtual processes;

FIG. 2 is a virtual process table;

FIG. 3 is a block diagram of a plurality of virtual processes;

FIG. 4 is a block diagram of a system for virtualizing resource ownership;

FIG. 5 is a block diagram of a system for virtualizing resource ownership;

FIG. 6 is a flowchart of a method for virtualizing resource ownership;

FIG. 7 is a block diagram of a system for virtualizing resource ownership;

FIG. 8 is a flowchart of a method for virtualizing resource ownership;

FIG. 9 is a block diagram of a system for virtualizing resource ownership;

FIG. 10 is a block diagram of a system for virtualizing resource ownership.

FIG. 11 is a block diagram of virtual processes and corresponding virtual super-users;

FIG. 12 is a block diagram of a system for virtualizing super-user privileges;

FIG. 13 is a block diagram of a system for virtualizing super-user privileges ;

FIG. 14 is a virtual super-user list; and

FIG. 15 is a flowchart of a method for virtualizing super-user privileges.

The Figures depict embodiments of the present invention for purposes of

illustration only. Those skilled in the art will readily recognize from the following

discussion that alternative embodiments of the illustrated structures and methods may be employed without departing from the principles of the invention described herein.

11/11/11 11:11:11

## DETAILED DESCRIPTION OF THE PREFERRED EMBODIMENTS

The present invention relates to virtualizing super-user privileges in a computer operating system including multiple virtual processes. One example of a virtual process is a virtual private server, which simulates a server running on a dedicated host machine.

As previously noted, implementing a virtual private server using traditional server technologies has been impossible because, rather than comprising a single, discrete process, a virtual private server must include a plurality of seemingly unrelated processes, each performing various elements of the sum total of the functionality required by a customer. Moreover, isolating virtual private servers from each other presents a number of difficulties related to resource ownership.

Accordingly, one aspect of the present invention relates to a system and method for associating identifiers with virtual processes, as described immediately below.

Thereafter, a system and method are described for virtualizing resource ownership in an computer operating system including multiple virtual processes. Finally, there is provided a detailed description of a system and method for virtualizing super-user privileges in a computer operating system including multiple virtual processes.

### I. Associating Identifiers with Virtual Processes

FIG. 1 is a high-level schematic block diagram of a system 100 for associating identifiers with virtual processes 101 according to one embodiment of the present

invention. A computer memory 102 includes a user address space 103 and an operating system address space 105. Multiple initialization processes 107 execute in the user address space 103. Although FIG. 1 illustrates only two initialization processes 107 executing in the user address space 103, those skilled in the art will understand that more than two initialization processes 107 can execute simultaneously within a given computer memory 102.

Also executing in the user address space 103 are one or more descendent processes 108 originating from the initialization processes 107. A descendent process 108 is a child process of an initialization process 107, or a child process thereof, extended to any number of generations of subsequent child processes. Although FIG. 1 illustrates only two descendent processes 108 for each initialization process 107, fewer or more than two descendent processes 108 per initialization process 107 can execute simultaneously within a given computer memory 102.

In one embodiment, a virtual process table 127 or other suitable data structure for storing associations 129 between executing processes 107, 108 and virtual processes 101 is inserted into the operating system 117. Of course, other data structures may be used to store associations 129, one example of which is a linked list.

In various embodiments, the virtual process table 127 (or other data structure) is dynamically loaded into the operating system kernel 109 while the kernel 109 is active. In another embodiment, the virtual process table 127 is stored in the user address space 103. The maintenance and use of the virtual process table 127 is discussed in detail below.

Those skilled in the art will recognize that a virtual process 101 is not an actual process that executes in the computer memory 102. Instead, the term "virtual process" describes a collection of associated functionality. For example, a virtual process 101 is not actually a discrete process, but instead, comprises a plurality of actual processes that together provide the desired functionality, thereby simulating the existence of a single application executing on a dedicated physical host. Each actual process that performs some of the functionality of the application is a part of the virtual process 101. As shown in FIG. 1, for example, initialization process 1 and descendent processes 1 and 2 together comprise one virtual process 101, whereas initialization process 2 and descendent processes 3 and 4 together comprise another.

As illustrated in FIG. 2, the virtual process table 127 stores, in one embodiment, an association 129 between a process identifier (PID) 201 and a virtual process identifier (VPID) 203. For example, the virtual process table 127 may store an association between a PID 201 of initialization process 1 (e.g., 3942) and a VPID 203 (e.g., 1). Likewise, an association 129b may be stored between a PID 201 of descendent process 1 (e.g., 6545), and the same VPID 203 (e.g., 1). Thus, initialization process 1 and descendent process 1 are said to be members of the same virtual process 101.

In order to associate a specific identifier with each actual process that is a member of a virtual process 101, a separate system initialization process 107 is started for each virtual process 101. Normally, each process executing on a multitasking operating system such as UNIX® is descended from a single system initialization process 107 that is started

when the operating system 117 is booted. However, the system 100 uses techniques described in detail below to start a separate system initialization process 107 for each virtual process 101. When each system initialization process 107 is started, an association 129 between the system initialization process 107 and the virtual process 101 is stored in the virtual process table 127. All additional processes that are descended from a given initialization process are thus identified with the virtual process 101 associated with that initialization process.

*See 107*  
In one embodiment, rather than starting a separate system initialization process 107 for each virtual process 101, a custom initialization process is started. In this embodiment, all processes that are members of a specific virtual process 101 are descended from the associated custom initialization process, and are associated with the virtual process 101 with which the custom initialization process is associated. The exact functionality included in the custom initialization process is a design choice that can be made by, for example, a system administrator.

System calls 115 that generate child processes (e.g., the UNIX<sup>®</sup> `fork()` and `clone()` functions) are intercepted so that the child processes can be associated with the virtual process 101 with which the parent process is associated. In one embodiment, a system call wrapper 111 is used to intercept system calls 115. In one embodiment, the wrapper 111 is dynamically loaded into the operating system kernel 109 while the kernel 109 is active. In another embodiment, the system call wrapper 111 is loaded into the user address space 103.

Pointers 114 to the system calls 115 are located in an operating system call vector table 113. Those skilled in the art will recognize that the term "system call vector table," as used herein, denotes an area in the operating system address space 105 in which addresses of system calls are stored. In the UNIX® operating system, this part of the operating system is called the "system call vector table," and that term is used throughout this description. Other operating systems employ different terminology to denote the same or similar system components. The pointers 114, themselves, are sometimes referred to as "system call vectors."

A copy 116 is made of a pointer 114 to each system call 115 to be intercepted. These copies 116 of pointers 114 may be stored in the operating system address space 105, but in an alternative embodiment, are stored in the user address space 103. Once the copies 116 have been made and saved, the pointers 114 in the system call vector table 113 to the system calls 115 to be intercepted are replaced with pointers 118 to the system call wrapper 111, such that when a system call 115 to be intercepted is made, the system call wrapper 111 executes instead.

In one embodiment, the system call wrapper 111 performs the process of copying, storing, and replacing of pointers. In other embodiments, the process of copying, storing, and replacing of pointers is performed by a pointer management module (not shown) executing in either the operating system address space 105 or the user address space 103, as desired. The pointer management module may either be a stand alone program or a component of a larger application program.



By intercepting a system call 115, alternative code is executed. The steps of inserting a system call wrapper 111 into the operating system 117, making a copy 116 of an operating system pointer 114 to a system call 115, and replacing the operating system pointer 114 with a pointer 118 to the system call wrapper 111 facilitate interception of a system call 115.

5 When a system call 115 to be intercepted is made, the operating system 117 uses the pointer 118 in the system call vector table 113 to the system call wrapper 111 to execute the system call wrapper 111.

10 In one embodiment, only the system calls 115 that create child processes need be intercepted, and thus only the pointers 114 to the system calls 115 to be intercepted are replaced with the pointers 118 to the system call wrapper 111. The pointers 114 to the system calls 115 which are not to be intercepted are not replaced. Thus, when a non-intercepted system call 115 is made, the actual system call 115 executes, not the system call wrapper 111.

15 The various initialization processes 107 and descendent processes 108 execute in the user address space 103 under control of the operating system 117 and make system calls 115. When a process makes a system call 115 that creates a child process, the system call wrapper 111 reads the virtual process table 127, and determines whether the process that made the system call (the parent of the child process being created) is associated with a virtual process 101. If so, the system call wrapper 111 uses the saved copy of the pointer  
20 116 to execute the system call 115, allowing the creation of the child process.

The system call wrapper 111 then updates the virtual process table 127, storing an association 129 between the newly created child process and the virtual process 101 with which the process that made the system call is associated. Thus, all descendent processes 108 are associated with the virtual process 101 with which their parent process is associated.

In one embodiment, the initialization processes 107 are started by a virtual process manager program 131 executing in the user address space 103. The virtual process manager program 131 modifies the operating system 117 of the computer to include the virtual process table 127. In one embodiment, the manager program 131 loads the virtual process table 127 into the kernel 109 of the operating system 117 while the kernel is active.

For each virtual process 101, the manager program 131 starts an initialization process 107 from which all other processes that are part of the virtual process 101 will originate as descendent processes 108. Each time the manager program 131 starts an initialization process 107 for a virtual process 101, the manager program 131 stores, in the virtual process table 127, an association 129 between the initialization process 107 and the appropriate virtual process 101. Subsequently, all additional processes that are part of the virtual process 101 will be originated from the initialization process, and thus associated with the appropriate virtual process 101.

For example, in this embodiment, the manager program 131 can start a first virtual process 101. To do so, the manager program 131 starts an initialization process 107 for the virtual process 101, storing an association 129 between the initialization process 107, and

a virtual process identifier for the virtual process 101. Additional processes that are part of the virtual process 101 originate from the initialization process 107, and are associated with the virtual process identifier of the virtual process 101. The manager program 131 can proceed to start a second virtual process 101 by starting a separate initialization process  
5 107, and associating the second initialization process 107 with a separate virtual process identifier for the second virtual process 101. Consequently, all of the processes associated with the second virtual process 101 will be associated with the appropriate virtual process identifier. In this manner, multiple virtual processes 101 on the same physical computer are each associated with unique identifiers.

10 In an alternative embodiment, the virtual process manager program 131 can be implemented as a modified loader program. A loader program is an operating system utility that is used to execute computer programs that are stored on static media. Typically, a loader program loads an executable image from static media into the user address space 103 of the computer memory 102, and then initiates execution of the loaded  
15 image by transferring execution to the first instruction thereof.

Like a standard loader program, a modified loader program loads executable images (in this case, initialization processes 107) from static media into the user address space 103. Additionally, the modified loader program stores, in the virtual process table  
20 127, an association 129 between the initialization process 107 being loaded and the appropriate virtual process 101. Thus, for each virtual process 101, an initialization process 107 is loaded by the modified loader program, and an association between the initialization

process 107 and the virtual process 101 is stored in the virtual process table 127. Subsequently, additional processes that are part of the virtual process 101 originate from the associated initialization process 107, and are thus associated with the virtual process 101, as described above.

5 In another embodiment, the modified loader program loads all processes that are part of each virtual process 101. In that embodiment, whenever the modified loader program loads a process, the modified loader program also stores, in the virtual process table 127, an association 129 between the loaded process and the appropriate virtual process 101.

## II. Virtualizing Resource Ownership

10 20 30 40 50 60 70 80 90 100 110 120 130 140 150 160 170 180 190 200 210 220 230 240 250 260 270 280 290 300 310 320 330 340 350 360 370 380 390 400 410 420 430 440 450 460 470 480 490 500 510 520 530 540 550 560 570 580 590 600 610 620 630 640 650 660 670 680 690 700 710 720 730 740 750 760 770 780 790 800 810 820 830 840 850 860 870 880 890 900 910 920 930 940 950 960 970 980 990 1000 1010 1020 1030 1040 1050 1060 1070 1080 1090 1100 1110 1120 1130 1140 1150 1160 1170 1180 1190 1200 1210 1220 1230 1240 1250 1260 1270 1280 1290 1300 1310 1320 1330 1340 1350 1360 1370 1380 1390 1400 1410 1420 1430 1440 1450 1460 1470 1480 1490 1500 1510 1520 1530 1540 1550 1560 1570 1580 1590 1600 1610 1620 1630 1640 1650 1660 1670 1680 1690 1700 1710 1720 1730 1740 1750 1760 1770 1780 1790 1800 1810 1820 1830 1840 1850 1860 1870 1880 1890 1900 1910 1920 1930 1940 1950 1960 1970 1980 1990 2000 2010 2020 2030 2040 2050 2060 2070 2080 2090 2100 2110 2120 2130 2140 2150 2160 2170 2180 2190 2200 2210 2220 2230 2240 2250 2260 2270 2280 2290 2300 2310 2320 2330 2340 2350 2360 2370 2380 2390 2400 2410 2420 2430 2440 2450 2460 2470 2480 2490 2500 2510 2520 2530 2540 2550 2560 2570 2580 2590 2600 2610 2620 2630 2640 2650 2660 2670 2680 2690 2700 2710 2720 2730 2740 2750 2760 2770 2780 2790 2800 2810 2820 2830 2840 2850 2860 2870 2880 2890 2900 2910 2920 2930 2940 2950 2960 2970 2980 2990 3000 3010 3020 3030 3040 3050 3060 3070 3080 3090 3100 3110 3120 3130 3140 3150 3160 3170 3180 3190 3200 3210 3220 3230 3240 3250 3260 3270 3280 3290 3300 3310 3320 3330 3340 3350 3360 3370 3380 3390 3400 3410 3420 3430 3440 3450 3460 3470 3480 3490 3500 3510 3520 3530 3540 3550 3560 3570 3580 3590 3600 3610 3620 3630 3640 3650 3660 3670 3680 3690 3700 3710 3720 3730 3740 3750 3760 3770 3780 3790 3800 3810 3820 3830 3840 3850 3860 3870 3880 3890 3900 3910 3920 3930 3940 3950 3960 3970 3980 3990 4000 4010 4020 4030 4040 4050 4060 4070 4080 4090 4100 4110 4120 4130 4140 4150 4160 4170 4180 4190 4200 4210 4220 4230 4240 4250 4260 4270 4280 4290 4300 4310 4320 4330 4340 4350 4360 4370 4380 4390 4400 4410 4420 4430 4440 4450 4460 4470 4480 4490 4500 4510 4520 4530 4540 4550 4560 4570 4580 4590 4600 4610 4620 4630 4640 4650 4660 4670 4680 4690 4700 4710 4720 4730 4740 4750 4760 4770 4780 4790 4800 4810 4820 4830 4840 4850 4860 4870 4880 4890 4900 4910 4920 4930 4940 4950 4960 4970 4980 4990 5000 5010 5020 5030 5040 5050 5060 5070 5080 5090 5100 5110 5120 5130 5140 5150 5160 5170 5180 5190 5200 5210 5220 5230 5240 5250 5260 5270 5280 5290 5300 5310 5320 5330 5340 5350 5360 5370 5380 5390 5400 5410 5420 5430 5440 5450 5460 5470 5480 5490 5500 5510 5520 5530 5540 5550 5560 5570 5580 5590 5600 5610 5620 5630 5640 5650 5660 5670 5680 5690 5700 5710 5720 5730 5740 5750 5760 5770 5780 5790 5800 5810 5820 5830 5840 5850 5860 5870 5880 5890 5900 5910 5920 5930 5940 5950 5960 5970 5980 5990 6000 6010 6020 6030 6040 6050 6060 6070 6080 6090 6100 6110 6120 6130 6140 6150 6160 6170 6180 6190 6200 6210 6220 6230 6240 6250 6260 6270 6280 6290 6300 6310 6320 6330 6340 6350 6360 6370 6380 6390 6400 6410 6420 6430 6440 6450 6460 6470 6480 6490 6500 6510 6520 6530 6540 6550 6560 6570 6580 6590 6600 6610 6620 6630 6640 6650 6660 6670 6680 6690 6700 6710 6720 6730 6740 6750 6760 6770 6780 6790 6800 6810 6820 6830 6840 6850 6860 6870 6880 6890 6900 6910 6920 6930 6940 6950 6960 6970 6980 6990 7000 7010 7020 7030 7040 7050 7060 7070 7080 7090 7100 7110 7120 7130 7140 7150 7160 7170 7180 7190 7200 7210 7220 7230 7240 7250 7260 7270 7280 7290 7300 7310 7320 7330 7340 7350 7360 7370 7380 7390 7400 7410 7420 7430 7440 7450 7460 7470 7480 7490 7500 7510 7520 7530 7540 7550 7560 7570 7580 7590 7600 7610 7620 7630 7640 7650 7660 7670 7680 7690 7700 7710 7720 7730 7740 7750 7760 7770 7780 7790 7800 7810 7820 7830 7840 7850 7860 7870 7880 7890 7900 7910 7920 7930 7940 7950 7960 7970 7980 7990 8000 8010 8020 8030 8040 8050 8060 8070 8080 8090 8100 8110 8120 8130 8140 8150 8160 8170 8180 8190 8200 8210 8220 8230 8240 8250 8260 8270 8280 8290 8300 8310 8320 8330 8340 8350 8360 8370 8380 8390 8400 8410 8420 8430 8440 8450 8460 8470 8480 8490 8500 8510 8520 8530 8540 8550 8560 8570 8580 8590 8600 8610 8620 8630 8640 8650 8660 8670 8680 8690 8700 8710 8720 8730 8740 8750 8760 8770 8780 8790 8800 8810 8820 8830 8840 8850 8860 8870 8880 8890 8900 8910 8920 8930 8940 8950 8960 8970 8980 8990 9000 9010 9020 9030 9040 9050 9060 9070 9080 9090 9100 9110 9120 9130 9140 9150 9160 9170 9180 9190 9200 9210 9220 9230 9240 9250 9260 9270 9280 9290 9300 9310 9320 9330 9340 9350 9360 9370 9380 9390 9400 9410 9420 9430 9440 9450 9460 9470 9480 9490 9500 9510 9520 9530 9540 9550 9560 9570 9580 9590 9600 9610 9620 9630 9640 9650 9660 9670 9680 9690 9700 9710 9720 9730 9740 9750 9760 9770 9780 9790 9800 9810 9820 9830 9840 9850 9860 9870 9880 9890 9900 9910 9920 9930 9940 9950 9960 9970 9980 9990 10000 10010 10020 10030 10040 10050 10060 10070 10080 10090 10100 10110 10120 10130 10140 10150 10160 10170 10180 10190 10200 10210 10220 10230 10240 10250 10260 10270 10280 10290 10300 10310 10320 10330 10340 10350 10360 10370 10380 10390 10400 10410 10420 10430 10440 10450 10460 10470 10480 10490 10500 10510 10520 10530 10540 10550 10560 10570 10580 10590 10600 10610 10620 10630 10640 10650 10660 10670 10680 10690 10700 10710 10720 10730 10740 10750 10760 10770 10780 10790 10800 10810 10820 10830 10840 10850 10860 10870 10880 10890 10900 10910 10920 10930 10940 10950 10960 10970 10980 10990 11000 11010 11020 11030 11040 11050 11060 11070 11080 11090 11100 11110 11120 11130 11140 11150 11160 11170 11180 11190 11200 11210 11220 11230 11240 11250 11260 11270 11280 11290 11300 11310 11320 11330 11340 11350 11360 11370 11380 11390 11400 11410 11420 11430 11440 11450 11460 11470 11480 11490 11500 11510 11520 11530 11540 11550 11560 11570 11580 11590 11600 11610 11620 11630 11640 11650 11660 11670 11680 11690 11700 11710 11720 11730 11740 11750 11760 11770 11780 11790 11800 11810 11820 11830 11840 11850 11860 11870 11880 11890 11900 11910 11920 11930 11940 11950 11960 11970 11980 11990 12000 12010 12020 12030 12040 12050 12060 12070 12080 12090 12100 12110 12120 12130 12140 12150 12160 12170 12180 12190 12200 12210 12220 12230 12240 12250 12260 12270 12280 12290 12300 12310 12320 12330 12340 12350 12360 12370 12380 12390 12400 12410 12420 12430 12440 12450 12460 12470 12480 12490 12500 12510 12520 12530 12540 12550 12560 12570 12580 12590 12600 12610 12620 12630 12640 12650 12660 12670 12680 12690 12700 12710 12720 12730 12740 12750 12760 12770 12780 12790 12800 12810 12820 12830 12840 12850 12860 12870 12880 12890 12900 12910 12920 12930 12940 12950 12960 12970 12980 12990 13000 13010 13020 13030 13040 13050 13060 13070 13080 13090 13100 13110 13120 13130 13140 13150 13160 13170 13180 13190 13200 13210 13220 13230 13240 13250 13260 13270 13280 13290 13300 13310 13320 13330 13340 13350 13360 13370 13380 13390 13400 13410 13420 13430 13440 13450 13460 13470 13480 13490 13500 13510 13520 13530 13540 13550 13560 13570 13580 13590 13600 13610 13620 13630 13640 13650 13660 13670 13680 13690 13700 13710 13720 13730 13740 13750 13760 13770 13780 13790 13800 13810 13820 13830 13840 13850 13860 13870 13880 13890 13900 13910 13920 13930 13940 13950 13960 13970 13980 13990 14000 14010 14020 14030 14040 14050 14060 14070 14080 14090 14100 14110 14120 14130 14140 14150 14160 14170 14180 14190 14200 14210 14220 14230 14240 14250 14260 14270 14280 14290 14300 14310 14320 14330 14340 14350 14360 14370 14380 14390 14400 14410 14420 14430 14440 14450 14460 14470 14480 14490 14500 14510 14520 14530 14540 14550 14560 14570 14580 14590 14600 14610 14620 14630 14640 14650 14660 14670 14680 14690 14700 14710 14720 14730 14740 14750 14760 14770 14780 14790 14800 14810 14820 14830 14840 14850 14860 14870 14880 14890 14900 14910 14920 14930 14940 14950 14960 14970 14980 14990 15000 15010 15020 15030 15040 15050 15060 15070 15080 15090 15100 15110 15120 15130 15140 15150 15160 15170 15180 15190 15200 15210 15220 15230 15240 15250 15260 15270 15280 15290 15300 15310 15320 15330 15340 15350 15360 15370 15380 15390 15400 15410 15420 15430 15440 15450 15460 15470 15480 15490 15500 15510 15520 15530 15540 15550 15560 15570 15580 15590 15600 15610 15620 15630 15640 15650 15660 15670 15680 15690 15700 15710 15720 15730 15740 15750 15760 15770 15780 15790 15800 15810 15820 15830 15840 15850 15860 15870 15880 15890 15900 15910 15920 15930 15940 15950 15960 15970 15980 15990 16000 16010 16020 16030 16040 16050 16060 16070 16080 16090 16100 16110 16120 16130 16140 16150 16160 16170 16180 16190 16200 16210 16220 16230 16240 16250 16260 16270 16280 16290 16300 16310 16320 16330 16340 16350 16360 16370 16380 16390 16400 16410 16420 16430 16440 16450 16460 16470 16480 16490 16500 16510 16520 16530 16540 16550 16560 16570 16580 16590 16600 16610 16620 16630 16640 16650 16660 16670 16680 16690 16700 16710 16720 16730 16740 16750 16760 16770 16780 16790 16800 16810 16820 16830 16840 16850 16860 16870 16880 16890 16900 16910 16920 16930 16940 16950 16960 16970 16980 16990 17000 17010 17020 17030 17040 17050 17060 17070 17080 17090 17100 17110 17120 17130 17140 17150 17160 17170 17180 17190 17200 17210 17220 17230 17240 17250 17260 17270 17280 17290 17300 17310 17320 17330 17340 17350 17360 17370 17380 17390 17400 17410 17420 17430 17440 17450 17460 17470 17480 17490 17500 17510 17520 17530 17540 17550 17560 17570 17580 17590 17600 17610 17620 17630 17640 17650 17660 17670 17680 17690 17700 17710 17720 17730 17740 17750 17760 17770 17780 17790 17800 17810 17820 17830 17840 17850 17860 17870 17880 17890 17900 17910 17920 17930 17940 17950 17960 17970 17980 17990 18000 18010 18020 18030 18040 18050 18060 18070 18080 18090 18100 18110 18120 18130 18140 18150 18160 18170 18180 18190 18200 18210 18220 18230 18240 18250 18260 18270 18280 18290 18300 18310 18320 18330 18340 18350 18360 18370 18380 18390 18400 18410 18420 18430 18440 18450 18460 18470 18480 18490 18500 18510 18520 18530 18540 18550 18560 18570 18580 18590 18600 18610 18620 18630 18640 18650 18660 18670 18680 18690 18700 18710 18720 18730 18740 18750 18760 18770 18780 18790 18800 18810 18820 18830 18840 18850 18860 18870 18880 18890 18900 18910 18920 18930 18940 18950 18960 18970 18980 18990 19000 19010 19020 19030 19040 19050 19060 19070 19080 19090 19100 19110 19120 19130 19140 19150 19160 19170 19180 19190 19200 19210 19220 19230 19240 19250 19260 19270 19280 19290 19300 19310 19320 19330 19340 19350 19360 19370 19380 19390 19400 19410 19420 19430 19440 19450 19460 19470 19480 19490 19500 19510 19520 19530 19540 19550 19560 19570 19580 19590 19600 19610 19620 19630 19640 19650 19660 19670 19680 19690 19700 19710 19720 19730 19740 19750 19760 19770 19780 19790 19800 19810 19820 19830 19840 19850 19860 19870 19880 19890 19900 19910 19920 19930 19940 19950 19960 19970 19980 19990 20000 20010 20020 20030 20040 20050 20060 20070 20080 20090 20100 20110 20120 20130 20140 20150 20160 20170 20180 20190 20200 20210 20220 20230 20240 20250 20260 20270 20280 20290 20300 20310 20320 20330 20340 20350 20360 20370 20380 20390 20400 20410 20420 20430 20440 20450 20460 20470 20480 20490 20500 20510 20520 20530 20540 20550 20560 20570 20580 20590 20600 20610 20620 20630 20640 20650 20660 20670 20680 20690 20700 20710 20720 20730 20740 20750 20760 20770 20780 20790 20800 20810 20820 20830 20840 20850 20860 20870 20880 20890 20900 20910 20920 20930 20940 20950 20960 20970 20980 20990 21000 21010 21020 21030 21040 21050 21060 21070 21080 21090 21100 21110 21120 21130 21140 21150 21160 21170 21180 21190 21200 21210 21220 21230 21240 21250 21260 21270 21280 21290 21300 21310 21320 21330 21340 21350 21360 21370 21380 21390 21400 21410 21420 21430 21440 21450 21460 21470 21480 21490 21500 21510 21520 21530 21540 21550 21560 21570 21580 21590 21600 21610 21620 21630 21640 21650 21660 21670 21680 21690 21700 21710 21720 21730 21740 21750 21760 21770 21780 21790 21800 21810 21820 21830 21840 21850 21860 21870 21880 21890 21900 21910 21920 21930 21940 21950 21960 21970 21980 21990 22000 22010 220

which permission has been granted by the owner). Thus, if a user attempts, for instance, to kill a process 301 that he or she does not own, the attempt fails and an error is generated.

A difficulty arises, however, in implementing resource ownership for multiple virtual processes 101 running on the same host system 300. Each virtual process 101 should be free to assign to an individual or group any UID 305 or GID 307, respectively. Indeed, some applications require certain processes 301 or files 303 to be associated with a particular UID 305 or GID 307 in order to properly function.

However, if two users of different virtual processes 101 share the same UID 305, those users could potentially kill each other's processes 301 and read, modify, or delete each other's files 303. The same is true for two groups sharing the same GID 307.

For instance, one user could execute a "kill -1" command, which terminates all of the processes 301 associated with the user's UID 305. Unfortunately, if another user on the same computer has the same UID 305, all of that user's processes 301 will be terminated as well. Clearly, this poses an unacceptable security risk and removes the appearance that the virtual process 101 is running on a dedicated physical host.

In accordance with the present invention, resource ownership is virtualized to allow a user of one virtual process 101 to appear to have the same UID 305 as a user of another virtual process 101, although neither user is capable of interfering with the processes 301 or accessing the files 303 of the other. Likewise, in accordance with the present invention, a group of users of one virtual process 101 may appear to share the same GID 307 with a group of users of another virtual process 101.

FIG. 4 illustrates a system 400 for virtualizing resource ownership. In one embodiment, a system call wrapper 111 intercepts a system call 115 for setting the UID 305 or GID 307 associated with a resource (such as a process 301 or file 303). In the case of UNIX®, for instance, the `setuid()` and `setgid()` functions are used to associate a UID 305 and GID 307, respectively, with a calling process 301. Similarly, the UNIX® `chown()` function is used to associate a UID 305 or GID 307 with a file 303. Of course, the invention is not restricted to any particular terminology or operating system.

A technique for intercepting system calls 115 was described above with reference to FIG. 1. As noted, pointers 114 to the system calls 115 to be intercepted can be copied and then replaced with pointers 118 to a system call wrapper 111. Thus, when the calls 115 are made, the system call wrapper 111 is executed instead.

For clarity, the following description often refers simply to the UID 305. However, the techniques and structures disclosed herein may also be used for system calls 115 involving GIDs 307, e.g., the UNIX® `setgid()` and `chown()` functions.

After the system call 115 is intercepted, the wrapper 111 determines a virtual process 101 corresponding to the calling process 301. The virtual process 101 is determined, in one implementation, by accessing the virtual process table 127, as described above, which stores associations 129 between processes 301 (e.g., PID 201) and virtual processes 101 (e.g., VPID 203).

Next, the wrapper 111 modifies the UID 305 specified in the intercepted call 115. In one implementation, the UID 305 is modified by encoding therein an indication of the

virtual process (e.g., VPID 203). For instance, in the case of Solaris®, a version of UNIX®, the UID 305 is a 32 bit word. In one embodiment, the UID 305 is divided into two 16 bit portions. As described in detail below, the VPID 203 is encoded within the upper 16 bits of the UID 305, while the lower 16 bits are used to store the original data from the UID 305.

5 In the illustrated embodiment, the VPID 203 is encoded within UID 305 according to the equation:

$$\text{UID} = \text{VPID} \ll 16 \mid \text{UID} \quad \text{Eq. 1}$$

where UID is the UID 305, VPID is the VPID 203 (from the table 127), and “<<” and “|” are the left shift and logical “OR” operators, respectively. In other words, the VPID 203 is left shifted 16 bits and then logically ORed with the UID 305.

Those skilled in the art will recognize that the above-described technique limits the number of unique UIDs 305 and virtual processes 101 to 65536, respectively. In alternative embodiments, however, the relative location and/or number of bits allocated to the VPID 203 within the UID 305 may vary, resulting in different limitations.

After the UID 305 is modified, the system call wrapper 111 associates the resource with the modified UID 305. This may be accomplished, in one embodiment, by executing the system call 115 by the wrapper 111, specifying the modified UID 305. In an alternative embodiment, the system call wrapper 111 can include its own code for setting the UID 305.

Consequently, from a standpoint of the calling process 301, the resource is  
20 associated with the UID 305 specified in the system call 115. From a standpoint of the

operating system 117, however, the resource is actually associated with the modified UID 305.

FIG. 4 provides an example of the above-described technique. Suppose that a process 301 having a PID 201 of 3942 attempts to execute the UNIX<sup>®</sup> `setuid()` system call 115 with a specified UID 305 of 1. As shown, the system call wrapper 111 uses the virtual process table 127 to determine the VPID 203 (e.g., 1) associated with the calling process 301. The VPID 203 is then encoded within UID 305 as described above, resulting in a modified UID 305 having a hexadecimal value of 0x00010001 (65537 in decimal). Accordingly, the calling process 301 is associated with a UID 305 of 65537 rather than the specified UID 305 of 1.

As shown in FIG. 5, a different UID 305 will result from a different VPID 203. For instance, suppose that the VPID 203 of the virtual process 101 of FIG. 5 has a value of 3. Applying the above-described equation, the resulting modified UID 305 has a hexadecimal value of 0x00030001 (196609 in decimal). Accordingly, the calling process 301 is associated with a UID 305 of 196609 rather than the original UID 305 of 1 or the modified UID 305 of 65537 from the previous example.

The above-described technique for virtualizing resource ownership is summarized in FIG. 6. A method 600 begins in one embodiment by loading 601 a system call wrapper 111 into the operating system 117. Thereafter, copies are made 603 of pointers 114 to selected system calls 115 to be intercepted (e.g., `setuid()`, `setgid()`, and `chown()`). The pointers 114 are then replaced 605, in one implementation, by pointers 118 to the



system call wrapper 111. Thus, when one of the selected system calls 115 is made, the system call wrapper 111 is executed instead.

A system call 115 for setting the UID 305 of a resource is then intercepted 607. Next, the system call wrapper 111 determines 609 the virtual process 101 corresponding to the calling process 301. In one embodiment, this determination is made by referencing the virtual process table 127, as described above.

After the virtual process 101 is determined, the system call wrapper 111 encodes 611 an indication of the virtual process 101 (e.g., the VPID 203) within the UID 305. The wrapper 111 then associates 613 the calling process 301 with the modified UID 305. In one implementation, this is accomplished by executing the system call 115 within the wrapper 111, specifying the modified UID 305.

Another aspect of virtualizing resource ownership involves intercepting system calls 115 for obtaining the UID 305 or GID 307 associated with a system resource. In the case of UNIX®, the `getuid()` function returns the UID 305 associated with the calling process 301. Similarly, the UNIX® `getgid()` function returns the GID 307. Additionally, the UNIX® `stat()` function returns the UID 305 and/or GID 307 associated with a file 303. Of course, the invention is not limited to any particular terminology or operating system 117.

Consequently, if a system call 115 for obtaining a UID 305 (e.g., `getuid()`) were allowed to execute without modification, the calling process 301 would receive a “modified” UID 305, such as a UID 305 including an indication of a virtual process 101.

From the standpoint of the calling process 301, the UID 305 would be unexpected, with unpredictable results.

Thus, FIG. 7 illustrates a system 700 for virtualizing resource ownership. After intercepting one of the above-identified system calls 115, the system call wrapper 111 obtains the UID 305 from the standpoint of the operating system 117. The wrapper 111 obtains the UID 305, in one embodiment, by executing the system call 115. In alternative embodiments, the wrapper 111 may include its own code for obtaining the UID 305.

In one embodiment, the UID 305 obtained by the wrapper 111 includes an indication of the virtual process 101 (e.g., VPID 203). Thus, the wrapper 111 removes the VPID 203 to restore the original, unmodified UID 305, as described in greater detail below.

As previously explained, a UID 305 in Solaris® is a 32 bit word. In one implementation, the upper 16 bits are used to encode the VPID 203, while the lower 16 bits are used to store the UID data. Thus, the VPID 203 may be removed from the UID 305 by applying the equation:

$$\text{UID} = 0x0000FFFF \& \text{UID} \quad \text{Eq. 2}$$

where UID is the UID 305 and "&" is the logical "AND" operator. In other words, the set of bits corresponding to the VPID 203 within the UID 305 are cleared. Of course, the encoding of the VPID 203 may vary in alternative embodiments, necessitating a different equation.

An example of the above-described process is shown in FIG 7. Suppose that a process 301 executes the UNIX® getuid() system call 115, which is intercepted by the

system call wrapper 111. The wrapper 111 obtains the UID 305 (e.g., 0x00010001) associated with the resource by executing, for example, the system call 115. As illustrated, the upper 16 bits of the UID 305 include an indication of a virtual process 101 (e.g., a VPID 203 of 1).

5 The wrapper 111 then removes the indication of the virtual process 101 by logically ANDing the UID 305 with a value configured to clear the bits associated with the VPID 203, (e.g., 65535). As a result, a UID 305 of 1 is returned to the calling process 301, rather than the UID 305 of 65537.

10 The above-described technique for virtualizing resource ownership is summarized in FIG. 8. A method 800 begins in one embodiment by loading 801 a system call wrapper 111 into the operating system 117. Thereafter, copies are made 803 of pointers 114 to selected system calls 115 to be intercepted (e.g., `getuid()`, `getgid()`, and `stat()`). The pointers 114 are then replaced 805, in one implementation, by pointers 118 to the system call wrapper 111. Thus, when one of the selected system calls 115 is made, the system call wrapper 111 is executed instead.

15 A system call 115 for obtaining the UID 305 associated with a resource is then intercepted 807. Next, the system call wrapper 111 obtains 809 the UID 305 associated with the resource. In one embodiment, the wrapper 111 obtains the UID 305 by executing the system call 115. As noted, the UID 305 includes, as a consequence of the method 600 of FIG. 6, an indication of a virtual process 101 (e.g., VPID 203).

20

After the UID 305 is obtained, the system call wrapper 111 removes 811 the VPID 203 by logically ANDing the UID 305 with an appropriate value, e.g., 65535. The UID 305 is then returned 813 to the calling process 301.

FIG. 9 illustrates an alternative system 900 for virtualizing resource ownership. In an alternative embodiment, an indication of the virtual process 101 is not encoded within the UID 305. Rather, after a system call 115 for setting a UID 305 is intercepted, the system call wrapper 111 selects an alternative UID 901 from a set 903 of available (unused) UIDs 305. The set 903 may be implemented using any suitable data structure, such as a table or linked list. The alternative UID 901 may be selected using any convenient method, such as selecting the next available UID 305 in the set 903.

Once the alternative UID 901 is selected, the wrapper 111 creates an association 905 in a translation data structure 907 between the UID 305 specified in the call 115, the alternative UID 901 selected by the wrapper 111, and an indication of the virtual process 101 (e.g., VPID 203), which may be obtained by the wrapper 111 from the virtual process table 127.

After the translation data structure 907 is updated, the wrapper 111 associates the calling process 301 with the alternative UID 901. This is accomplished, in one embodiment, by executing the system call 115, specifying the alternative UID 901.

FIG. 9 provides an example of the above-described technique. Suppose that a process 301 having a PID 201 of 3942 attempts to execute the UNIX<sup>®</sup> `setuid()` system call 115 with a specified UID 305 of 1. As illustrated, the system call wrapper 111 intercepts the

call 115 and uses the virtual process table 127 to determine the virtual process 101 (e.g., VPID 203) associated with the calling process 301.

The system call wrapper 111 then selects an alternative UID 901 (e.g., 1003) from a set 903 of available UIDs 305. Thereafter, the wrapper 111 creates an association 905 in the translation data structure 907 between the UID 305 specified in the call 115 (e.g., 1), the alternative UID 901 (e.g., 1003), and the VPID 203 (e.g., 1). Once the translation data structure 907 is updated, the wrapper 111 associates the calling process 301 with the alternative UID 901 by executing, for example, the system call 115.

FIG. 10 illustrates a corresponding system 1000 for intercepting system calls 115 for obtaining the UID 305 or GID 307 associated with a resource. Initially, the system call wrapper 111 intercepts the call 115 (e.g., `getuid()`, `getgid()`, and `stat()`). Thereafter, the wrapper 111 determines the virtual process 101 (e.g., VPID 203) associated with the calling process 301 using a virtual process table 127 or the like.

The system call wrapper 111 then obtains the alternative UID 901 associated with the resource by executing, for example, the system call 115. As described above, the alternative UID 901 is associated with the resource as a consequence of the system 900 illustrated in FIG. 9.

After the alternative UID 901 is obtained, the wrapper 111 accesses the translation data structure 907, looking up the alternative UID 901 and the VPID 203. When an association 905 is found, the corresponding UID 305 is retrieved from the translation data structure 907 and returned to the calling process 301.

An example of the above-described process is shown in FIG. 10. Suppose that a process 301 executes the `getuid()` function, which is intercepted by the system call wrapper 111. In one embodiment, the wrapper 111 executes the `getuid()` function, which returns an alternative UID 901 of 1003. The wrapper 111 also determines the VPID 203 (e.g., 2) associated with the calling process 301 by accessing the virtual process table 127.

5 127. The wrapper 111 then accesses the translation data structure 907, looking up a UID 901 of 1003 and a VPID 203 of 2. As illustrated, an association 905 exists, revealing a UID 305 of 1, which is subsequently returned to the calling process 301.

### III. Virtualizing Super-User Privileges

As noted above, in UNIX® and related operating systems, the “super-user” is granted special privileges not available to other users. For example, the super-user can open, modify, or delete the files of other users, as well as terminate other users’ processes. Indeed, the super-user can add and delete users, assign and change passwords, and insert modules into the operating system kernel 109.

Implementing super-user privileges in an operating system 117 including multiple virtual processes 101 presents numerous challenges. For example, each virtual process 101 should be allowed to have a user who is granted super-user-like powers, 20 e.g., the ability to add and delete users of the virtual process 101, access files 303 of any

user of the virtual process 101, terminate processes 301 associated with the virtual process 101, and the like.

However, if a user of each virtual process 101 was given full super-user privileges, a super-user of one virtual process 101 could access the files 303 of a user of another virtual process 101. Similarly, a super-user of one virtual process 101 could terminate the processes 301 associated with a user of another virtual process. 101 Indeed, a super-user of one virtual process 101 could obtain exclusive access to all system resources, effectively disabling the other virtual processes 101. Clearly, granting a user of each virtual process 101 full super-user privileges would seriously compromise system security, entirely removing the illusion that each virtual process 101 is running on a dedicated host computer.

As illustrated in FIG. 11, the present invention solves the foregoing problems, in one embodiment, by designating a plurality of virtual super-users 1101, typically one per virtual process 101. A virtual super-user 1101 has many of the privileges of an actual super-user with respect to his or her own virtual process 101. For example, a virtual super-user 1101 can add and delete users of the virtual process 101, access files 303 of any user of the virtual process 101, terminate processes 301 associated with the virtual process 101, and the like. However, a virtual super-user 1101 cannot, for instance, add or delete users of other virtual processes 101, access the files 303 of users of other virtual processes 101, or terminate the processes 301 associated with other virtual processes 101.





From the standpoint of the operating system 117, however, the VSUID 1103 is not a super-user UID 305, and does not convey any super-user privileges. For example, a VSUID 1103 of 0x00010000 has a decimal value of 65536, clearly not a UID 305 of zero. Thus, without more, a virtual super-user 1101 would have all of the limitations of a regular user.

Consequently, as shown in FIG. 12, selected system calls 115 are intercepted for performing operations requiring actual super-user privileges, which are nevertheless desirable for a virtual super-user 1101 to perform in the context of his or her own virtual process 101. For example, system calls 115 are intercepted that operate on files 303, e.g., `open()`, `creat()`, `link()`, `unlink()`, `chdir()`, `fchdir()`, `symlink()`, `readlink()`, `readdir()`, `access()`, `rename()`, `mkdir()`, `rmdir()`, `truncate()`, and `ftruncate()`. Of course, those skilled in the art will recognize that the invention is not limited to any particular operating system 117 or terminology.

As noted above, a normal user is typically restricted from opening, deleting, renaming, etc., a file 301 owned by another user. However, a virtual super-user 1101 should appear, in most respects, to be an actual super-user for operations pertaining to his or her own virtual process 101.

Thus, in one embodiment, if a system call 115 is "made" by a virtual super-user 1101 (i.e. by a process 301 owned by a virtual super-user 1101) and pertains to the virtual process 101 of the virtual super-user 1101, then actual super-user privileges are temporarily granted to the virtual super-user 1101 for purposes of the system call 115.

This may be accomplished, in one embodiment, by executing an appropriate function, e.g. `setuid()`, to assign a UID 305 of zero or other designation of super-user privileges to the calling process 301. After the system call 115 is executed, the super-user privileges may be withdrawn by executing the same function to restore the VSUID 1103.

Whether the system call 115 was made by a virtual super-user 1101 may be determined by checking whether the owner of the calling process 301 has a VSUID 1103. Of course, if the system call 115 was not made by a virtual super-user 1101, the wrapper 111 preferably disallows execution of the system call 111. For instance, the wrapper 111 may generate an error message, indicating a privilege violation. Alternatively, the wrapper 111 may simply allow the system call 115 to proceed without granting actual super-user privileges, resulting in the operating system 117 disallowing execution of the system call 115, since the VSUID 1103 does not convey actual super-user privileges.

Whether the system call 115 pertains to the virtual process 101 of the virtual super-user 1101 may be determined by checking whether the system resource(s) affected by the system call 115 relate to the virtual process 101 of the virtual super-user 1101. For example, with respect to system calls 115 that affect processes 301 (such as `kill()`), the virtual process table 127 may be checked to determine whether the process 301 has an association 129 with the virtual process 101 of the virtual super-user 1101. Similarly, in one embodiment, each virtual process 101 has a distinct file system,

allowing the wrapper 111 to easily determine whether a file 303 referenced by the call 115 is associated with the virtual process 101 of the virtual super-user 1101.

As shown in FIG. 12, suppose that a process 301 owned by a virtual super-user 1101 attempts to execute the open ( ) system call 115 in order to open another user's file 303, which is nevertheless associated with the virtual process 101 of the virtual super-user 1101. The virtual process 101 (e.g., VPID 203) may be determined, in one embodiment, by referencing the virtual process table 127 using the PID 201 of "3542."

Since the file 303 pertains to the virtual process 101 of the virtual super-user 1101, the system call wrapper 111 temporarily grants actual super-user privileges to the virtual super-user 1101. In the illustrated embodiment, this is accomplished by executing an appropriate system call 1201 (e.g., in UNIX®, the setuid ( ) function with a UID 305 of zero). The system call 115 is then executed, after which the wrapper 111 withdraws the actual super-user privileges 1101 by executing, for example, an appropriate system call 1203 (e.g., in UNIX®, the setuid ( ) function with the original VSUID 1103 of the virtual super-user 1101). This approach grants super-user privileges on a call-by-call basis.

Thus, a virtual super-user 1101 may perform an operation for which actual super-user privileges are required, without granting the virtual super-user 1101 unfettered access to all of the system's resources. This allows each virtual process 101 to have at least one system administrator with limited super-user-like powers, while

maintaining the illusion that each virtual process 101 is running on a dedicated host computer.

Other system calls 115 that may be intercepted include system calls 115 for terminating a process 301. In UNIX®, the `kill()` system call 115 allows a user to terminate one or more processes 301. For example, executing the `kill()` system call 115 with a specified process 301 (e.g., PID 201) terminates that process 301. Executing the `kill()` system call 115 with an argument of -1 results in the termination of all of the user's processes 301. An argument of less than -1 results in the termination of all of the processes 301 associated with a group (e.g., GID 307 taken from the absolute value of the argument).

As noted above, a super-user may terminate any system process 301. Thus, if the super-user specifies a PID 201, the corresponding process 301 will be terminated.

Likewise, if the super-user specifies a negative GID 307, the processes 301 belonging to the specified group are terminated. If, however, the super-user specifies an argument of -1, all processes 301 other than those with PID 201 of 0 or 1 are terminated.

In one embodiment, it is desirable for a virtual super-user 1101 to be able to terminate processes 301 associated with his or her virtual process 101. Accordingly, the system call wrapper 111 intercepts system calls 115 for terminating processes 301 (e.g., `kill()`).

Where a virtual super-user 1101 attempts to terminate a specific process 301 associated with his or her virtual process 101, the wrapper 111 proceeds as discussed

above with reference to FIG. 12. In other words, the wrapper 111 grants temporary actual super-user privileges 101 to the calling process 301 and allows execution of the system call 115.

However, as shown in FIG. 13, where the system call 115 specifies a negative parameter, the wrapper 111 proceeds differently. Since the powers of virtual super-user 1101 should be limited to his or her virtual process 101, a `kill()` system call 115 with an argument of -1 results only in the termination of processes 101 associated with the virtual process 101. Thus, in one embodiment, a `kill(-1)` system call 115 "pertains" to the virtual process 101 by definition.

In one embodiment, the system call wrapper 111 iterates through the virtual process table 127, terminating all processes 301 associated with the virtual process 101. Thus, a `kill(-1)` system call 115 operates in the manner expected, maintaining the illusion that the virtual process 101 of the virtual super-user 1101 is executing on a dedicated host machine.

80 Likewise, in the case of argument of less than -1, denoting a GID 307, the wrapper 111 cycles through all of the processes 301 associated with the virtual process 101 of the virtual super-user 1101 and determines whether each such process 301 corresponds the specified group (e.g., GID 307). If so, those processes 301 are terminated in the manner discussed above.

20 As an example, as shown in FIG. 13, suppose that a process 301 is associated virtual process 1 (e.g., having a VPID 203 of 1). The process 301 is owned by a virtual

super-user 1101 by virtue of the VSUID 1103 (e.g., 0x00010000), and pertains to the virtual process 101 by definition. Accordingly, the wrapper 111 grants temporary actual super-user privileges to the calling process 301 by executing the system call 1201.

Thereafter, the wrapper 111 iterates through the virtual process table 127, identifying each process 301 (e.g., PIDs 3942 and 4400) associated with a VPID 203 of 1. System calls 115 (e.g., `kill(3942)`, `kill(4400)`) are then made to terminate each of the identified processes 301, after which the actual super-user privileges are withdrawn by executing the system call 1203.

A variety of other system calls 115 may be intercepted within the scope of the invention in order to grant limited super-user privileges to a virtual super-user 1101. Those skilled in the art will know how to apply the above-described techniques in the context of these other system calls 115.

In some instances, it is desirable to prevent a virtual super-user 1101 from executing certain system calls 115 altogether. For example, in UNIX®, the `insmod()` and `rmmmod()` functions allow a super-user to insert modules into, and remove modules from, the operating system kernel 109. Giving such powers to a virtual super-user 1101 could seriously compromise system security, allowing the virtual super-user 1101 to alter the basic functionality of the operating system 117.

In one embodiment, a virtual super-user 1101 is prevented from executing a system calls 115 for which actual super-user privileges are required by simply not intercepting the call 115. Since the VSUID 1103 is not a super-user UID 305, the

operating system 115 will automatically reject an attempt by a virtual super-user 1101 to execute, for example, the `insertmod()` call 115.

In an alternative embodiment of the invention, a virtual super-user 1101 is not designated by assigning a VSUID 1103, as discussed above. Rather, a virtual super-user 1101 is simply assigned a UID 305 as in the case of other users. Thereafter, the assigned UID 305 is stored in a virtual super-user list 1401 or other suitable data structure, as illustrated in FIG. 14, together with an indication of the virtual process 101 (e.g., VPID 203). Accordingly, when selected system calls 115 are intercepted for which actual super-user privileges are required, a user may be identified as a virtual super-user 1101 by consulting the virtual super-user list 1401.

Since virtual super-users 1101 in this embodiment are given regular UIDs 305, the possibility of conflicts between virtual processes 101 arises. However, such conflicts may be resolved using the techniques described in FIGS. 9-10, i.e. intercepting system calls 115 for setting a UID 305 of a resource and assigning an alternative UID 901. Thus, virtual super-users 1101 of different virtual processes 101 may appear to share the same UID 305 without conflict.

FIG. 15 summarizes the above-described techniques. A method 1500 for virtualizing super-user privileges has two phases, preparation and operation. The preparation phase begins by loading 1501 a system call wrapper 111 into the operating system 117. Thereafter, copies are made 1503 of pointers 114 to selected system calls 115 for performing operations for which actual super-user privileges are required,

which are nevertheless desirable to be performed by a virtual super-user 1101 with respect to his or her own virtual process 101 (e.g., open(), kill(), etc.). The pointers 114 are then replaced 1505, in one implementation, by pointers 118 to the system call wrapper 111. Thus, when one of the selected system calls 115 is made, the system call  
5 wrapper 111 is executed instead

During the operation phase, a system call 115 is intercepted 1507 by the system call wrapper 111. Thereafter, the wrapper 111 determines 1509 whether the call 115 was "made" by a virtual super-user 1101 (i.e. by a process 301 owned by a virtual super-user 1101). If not, the system call 115 is disallowed 1511, and the method 1500 ends.

10 If, however, the call 115 was made by a virtual super-user 1101, a determination 1513 is made whether the call 115 pertains to the virtual process 101 of the virtual super-user 1101. If not, the call 115 is disallowed, and the method 1500 ends.

15 If, however, the call 115 pertains to the virtual process 101 of the virtual super-user 1101, actual super-user privileges are granted to the virtual super-user, after which the system call 115 is executed 1517. Finally, the actual super-user privileges are withdrawn 1519, and the method 1500 ends.

20 *See 0.19.7* If view of the foregoing, the present invention offers numerous advantages not available in conventional approaches. For example, super-user privileges are virtualized in an operating system 117 including multiple virtual processes 101, such that a virtual super-user has the power to perform traditional system administrator functions with respect to his or her own virtual process 101, but is unable to interfere



with other virtual processes 101 or the underlying operating system 117. Thus, each virtual process 101 can have a virtual super-user 1101, while preserving the illusion that the virtual processes 101 are running on dedicated host machines.

As will be understood by those familiar with the art, the invention may be embodied in other specific forms without departing from the spirit or essential characteristics thereof. Likewise, the particular naming of the modules, features, attributes or any other aspect is not mandatory or significant, and the mechanisms that implement the invention or its features may have different names or formats.

Accordingly, the disclosure of the present invention is intended to be illustrative, but not limiting, of the scope of the invention, which is set forth in the following claims.